ANT - QUICK GUIDE

Apache Ant is a Java based build tool from Apache Software Foundation. Apache Ant's build files are written in XML and take advantage of the open standard, portable and easy to understand nature of XML.

Why do you need a build tool?

Before diving deep into the definition of Apache Ant, one must understand the need for a build tool. Why do I need Ant, or more specifically, why do I need a build tool?

Do you spend your day doing the following manually?

- Compile code
- Package the binaries
- Deploy the binaries to the test server
- Test your changes
- Copy code from one location to another

If you have answered yes to any of the above, then it is time to automate the process and take away that burden from you.

On average, a developer spends 3 hours (out of a 8 hour working day) doing mundane tasks like build and deployment. Wouldn't you be delighted to get back the 3 hours?

Enter Apache Ant. Apache Ant is an operating system build and deployment tool that can be executed from a command line.

Features of Apache Ant

- Ant is the most complete Java build and deployment tool available.
- Ant is platform neutral and can handle platform specific properties such as file separators.
- Ant can be used to perform platform specific tasks such as modifying the modified time of a file using 'touch' command.
- Ant scripts are written using plain XML. If you are already familiar with XML, you can learn Ant pretty quickly.
- Ant is good at automating complicated repetitive tasks.
- Ant comes with a big list of predefined tasks.
- Ant provides an interface to develop custom tasks.
- Ant can be easily invoked from the command line and it can integrate with free and commercial IDEs.

Installing Apache Ant

It is assumed that you have already downloaded and installed Java Development Kit (JDK) on your computer. If not, please follow the instructions here.

Apache Ant is distributed under the Apache Software License, a fully-fledged open source license certified by the open source initiative.

The latest Apache Ant version, including full-source code, class files and documentation can be found at http://ant.apache.org.

- Ensure that the JAVA_HOME environment variable is set to the folder where your JDK is installed.
- Download the binaries from http://ant.apache.org
- Unzip the zip file to a convenient location using Winzip, winRAR, 7-zip or similar tools, say **c:** folder.
- Create a new environment variable called **ANT_HOME** that points to the Ant installation folder, in this case **c:\apache-ant-1.8.2-bin** folder.
- Append the path to the Apache Ant batch file to the PATH environment variable. In our case this would be the **c:\apache-ant-1.8.2-bin\bin** folder.

Ant - Build Files

Typically, Ant's build file, **build.xml** should live in the project's base directory. Although you are free to use other file names or place the build file in some other location.

For this exercise, create a file called build.xml anywhere in your computer.

```
<?xml version="1.0"?>
   <project name="Hello World Project" default="info">
   <target name="info">
      <echo>Hello World - Welcome to Apache Ant!</echo>
   </target>
</project>
```

Please note that there should be no blank lines or whitespaces before the xml declaration. If you do, this may cause an error message when running the ant build - *The processing instruction target matching "[xX][mM][lL]" is not allowed.*

All buildfiles require the **project** element and at least one **target** element.

The XML element **project** has three attributes :

Attributes	Description
name	The Name of the project. (Optional)
default	The default target for the build script. A project may contain any number of targets. This attribute specifies which target should be considered as the default. (Mandatory)
basedir	The base directory (or) the root folder for the project. (Optional)

A target is a collection of tasks that you want to run as one unit. In our example, we have a simple target to provide an informational message to the user.

Targets can have dependencies on other targets. For example, a **deploy** target may have a dependency on the **package** target, and the **package** target may have a dependency on the **compile** target and so forth. Dependencies are denoted using the **depends** attribute. For example:

```
<target name="deploy" depends="pacakge">
  ....
</target>
<target name="pacakge" depends="clean,compile">
  ....
</target>
<target name="clean" >
  ....
</target>
<target name="compile" >
  ....
</target>
```

The target element has the following attributes:

Attributes	Description
name	The name of the target (Required)
depends	Comma separated list of all targets that this target depends on. (Optional)
description	A short description of the target. (optional)
if	Allows the execution of a target based on the trueness of a conditional attribute. (optional)
unless	Adds the target to the dependency list of the specified Extension Point. An Extension Point is similar to a target, but it does not have any tasks. (Optional)

Ant - Property Task

Ant build files are written in XML, which does not cater for declaring variables as you do in your favourite programming language. However, as you may have imagined, it would be useful if Ant allowed declaring variables such as project name, project source directory etc.

Ant uses the **property** element which allows you to specify properties. This allows the properties to be changed from one build to another. or from one environment to another.

By default, Ant provides the following pre-defined properties that can be used in the build file

Properties	Description
ant.file	The full location of the build file.
ant.version	The version of the Apache Ant installation.
basedir	The basedir of the build, as specified in the **basedir** attribute of the **project** element.
ant.java.version	The version of the JDK that is used by Ant.
ant.project.name	The name of the project, as specified in the **name** atrribute of the **project** element
ant.project.default-target	The default target of the current project
ant.project.invoked-targets	Comma separated list of the targets that were invoked in the current project

ant.core.lib	The full location of the ant jar file
ant.home	The home directory of Ant installation
ant.library.dir	The home directory for Ant library files - typically ANT_HOME/lib folder.

Ant also makes the system properties (Example: file.separator) available to the build file.

In addition to the above, the user can define additional properties using the **property** element. An example is presented below which shows how to define a property called **sitename**:

```
<?xml version="1.0"?>
<project name="Hello World Project" default="info">
  <property name="sitename" value="www.tutorialspoint.com"/>
   <target name="info">
   <echo>Apache Ant version is ${ant.version} - You are
   at ${sitename} </echo>
   </target>
</project>
```

Running ant on the above build file should produce the following output:

```
C:\>ant
Buildfile: C:\build.xml

info:
     [echo] Apache Ant version is Apache Ant(TM) version 1.8.2
     compiled on December 20 2010 - You are at www.tutorialspoint.com

BUILD SUCCESSFUL
Total time: 0 seconds
C:\>
```

Ant - Property Files

Setting properties directly in the build file is okay if you are working with a handful of properties. However, for a large project, it makes sense to store the properties in a separate property file.

Storing the properties in a separate file allows you to reuse the same build file, with different property settings for different execution environment. For example, build properties file can be maintained separately for DEV, TEST and PROD environments.

Specifying properties in a separate file is useful when you do not know the values for a property (in a particular environment) up front. This allows you to perform the build in other environments where the property value is known.

There is no hard and fast rule, but typically the property file is named **build.properties** and is placed along side the **build.xml** file. You could create multiple build properties file based on the deployment environments - such as build.properties.dev and build.properties.test

The contents of the build property file are similar to the normal java property file. They contain one property per line. Each property is represented by a name and a value pair. The name and value pair are separated by an equals sign. It is highly recommended that the properties are annotated with proper comments. Comments are listed using the has character.

The following shows a **build.xml** and an associated **build.properties** file

build.xml

```
<?xml version="1.0"?>
```

```
<project name="Hello World Project" default="info">
   <property file="build.properties"/>
      <target name="info">
         <echo>Apache Ant version is ${ant.version} - You are
         at ${sitename} </echo>
     </target>
</project>
```

build.properties

```
# The Site Name
sitename=www.tutorialspoint.com
buildversion=3.3.2
```

Ant - Data Types

Ant provides a number of predefined data types. Do not confuse the data types that are available in the programming language, but instead consider the data types as set of services that are built into the product already

The following is a list of data types provided by Apache Ant

File Set

The Fileset data types represents a collection of files. The Fileset data type is usually used as a filter to include and exclude files that match a particular pattern.

For example:

```
<fileset dir="${src}" casesensitive="yes">
  <include name="**/*.java"/>
  <exclude name="**/*Stub*"/>
</fileset>
```

The **src** attribute in the above example points to the source folder of the project.

In the above example, the fileset selects all java files in the source folder except those that contain the word 'Stub' in them. The casesensitive filter is applied to the fileset which means that a file with the name **Samplestub.java** will not be excluded from the fileset

Pattern Set

A pattern set is a pattern that allows to easily filter files or folders based on certain patterns. Patterns can be created using the following meta characters.

- **?** - Matches one character only
- ***** - Matches zero or many characters
- ****** - Matches zero or many directories recursively

The following example should give an idea of the usage of a pattern set.

```
<patternset >
  <include name="src/**/*.java"/>
  <exclude name="src/**/*Stub*"/>
</fileset>
```

The patternset can then be reused with a fileset as follows:

```
<fileset dir="${src}" casesensitive="yes">
```

```
  <patternset ref/>
 </fileset>
```

File List

The File list data type is similar to the file set except that the File List contains explicitly named lists of files and do not support wild cards

Another major difference between the file list and the file set data type is that the file list data type can be applied for files that may or may not exist yet.

Following is an example of the File list data type

```
<filelist >
  <file name="applicationConfig.xml"/>
  <file name="faces-config.xml"/>
  <file name="web.xml"/>
  <file name="portlet.xml"/>
</filelist>
```

The **webapp.src.folder** attribute in the above example points to the web application's source folder of the project.

Filter Set

Using a Filter Set data type with the copy task, you can replace certain text in all files that match the pattern with a replacement value.

A common example is to append the version number to the release notes file, as shown in the example below

```
<copy todir="${output.dir}">
  <fileset dir="${releasenotes.dir}" includes="**/*.txt"/>
  <filterset>
    <filter token="VERSION" value="${current.version}"/>
  </filterset>
</copy>
```

The **output.dir** attribute in the above example points to the output folder of the project.

The **releasenotes.dir** attribute in the above example points to the release notes folder of the project.

The **current.version** attribute in the above example points to the current version folder of the project.

The copy task, as the name suggests is used to copy files from one location to another.

Path

The **path** data type is commonly used to represent a classpath. Entries in the path are separated using a semicolon or colon. However, these characters are replaced a the run time by the running system's path separator character.

Most commonly, the classpath is set to the list of jar files and classes in the project, as shown in the example below:

```
<path >
  <pathelement path="${env.J2EE_HOME}/${j2ee.jar}"/>
  <fileset dir="lib">
      <include name="**/*.jar"/>
  </fileset>
</path>
```

The **env.J2EE_HOME** attribute in the above example points to the environment variable **J2EE_HOME**.

The **j2ee.jar** attribute in the above example points to the name of the J2EE jar file in the J2EE base folder.

Ant - Building Projects

Now that we have learnt about the data types in Ant, it is time to put that into action. Consider the following project structure

This project will form the **Hello World** project for the rest of this tutorial.

```
C:\work\FaxWebApplication>tree
Folder PATH listing
Volume serial number is 00740061 EC1C:ADB1
C:.
+---db
+---src
.   +---faxapp
.       +---dao
.       +---entity
.       +---util
.       +---web
+---war
    +---images
    +---js
    +---META-INF
    +---styles
    +---WEB-INF
        +---classes
        +---jsp
        +---lib
```

Let me explain the project structure.

- The database scripts are stored in the **db** folder.
- The java source code is stored in the **src** folder.
- The images, js, META-INF, styles (css) are stored in the **war** folder.
- The JSPs are stored in the **jsp** folder.
- The third party jar files are stored in the **lib** folder.
- The java class files will be stored in the **WEB-INF\classes** folder.

The aim of this exercise is to build an ant file that compiles the java classes and places them in the WEB-INF\classes folder.

Here is the build.xml required for this project. Let us consider it piece by piece

```
<?xml version="1.0"?>
<project name="fax" basedir="." default="build">
    <property name="src.dir" value="src"/>
    <property name="web.dir" value="war"/>
    <property name="build.dir" value="${web.dir}/WEB-INF/classes"/>
    <property name="name" value="fax"/>

    <path >
        <fileset dir="${web.dir}/WEB-INF/lib">
            <include name="*.jar"/>
        </fileset>
        <pathelement path="${build.dir}"/>
    </path>

    <target name="build" description="Compile source tree java files">
        <mkdir dir="${build.dir}"/>
        <javac destdir="${build.dir}" source="1.5" target="1.5">
```

```
            <src path="${src.dir}"/>
            <classpath ref/>
        </javac>
    </target>

    <target name="clean" description="Clean output directories">
        <delete>
            <fileset dir="${build.dir}">
                <include name="**/*.class"/>
            </fileset>
        </delete>
    </target>
</project>
```

First, let us declare some properties for the source, web and build folders.

```
<property name="src.dir" value="src"/>
<property name="web.dir" value="war"/>
<property name="build.dir" value="${web.dir}/WEB-INF/classes"/>
```

In this example, the **src.dir** refers to the source folder of the project (i.e, where the java source files can be found).

The **web.dir** refers to the web source folder of the project. This is where you can find the JSPs, web.xml, css, javascript and other web related files

Finally, the **build.dir** refers to the output folder of the project compilation.

Properties can refer to other properties. As shown in the above example, the **build.dir** property makes a reference to the **web.dir** property.

In this example, the **src.dir** refers to the source folder of the project.

The default target of our project is the **compile** target. But first let us look at the **clean** target.

The clean target, as the name suggests deletes the files in the build folder.

```
<target name="clean" description="Clean output directories">
   <delete>
      <fileset dir="${build.dir}">
         <include name="**/*.class"/>
      </fileset>
   </delete>
</target>
```

The master-classpath holds the classpath information. In this case, it includes the classes in the build folder and the jar files in the lib folder.

```
<path >
   <fileset dir="${web.dir}/WEB-INF/lib">
      <include name="*.jar"/>
   </fileset>
   <pathelement path="${build.dir}"/>
</path>
```

Finally, the build target to build the files. First of all, we create the build directory if it doesn't exist. Then we execute the javac command (specifying jdk1.5 as our target compilation). We supply the source folder and the classpath to the javac task and ask it to drop the class files in the build folder.

```
<target name="build" description="Compile main source tree java files">
   <mkdir dir="${build.dir}"/>
   <javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
             deprecation="false" optimize="false" failonerror="true">
      <src path="${src.dir}"/>
```

```
      <classpath ref/>
   </javac>
</target>
```

Running ant on this file will compile the java source files and place the classes in the build folder.

The following outcome is the result of running the ant file:

```
C:\>ant
Buildfile: C:\build.xml

BUILD SUCCESSFUL
Total time: 6.3 seconds
```

The files are compiled and are placed in the **build.dir** folder.

Ant - Creating JAR files

The next logical step after compiling your java source files, is to build the java archive, i,e the JAR file. Creating JAR files with Ant is quite easy with the **jar** task. Presented below are the commonly used attributes of the jar task

Attributes	Description
basedir	The base directory for the output JAR file. By default, this is set to the base directory of the project.
compress	Advises ant to compress the file as it creates the JAR file.
keepcompression	While the **compress** attribute is applicable to the individual files, the **keepcompression** attribute does the same thing, but it applies to the entire archive.
destfile	The name of the output JAR file
duplicate	Advises Ant on what to do when duplicate files are found. You could add, preserve or fail the duplicate files.
excludes	Advises Ant to not include these comma separated list of files in the package.
excludesfile	Same as above, except the exclude files are specified using a pattern.
inlcudes	Inverse of excludes
includesfile	Inverse of excludesfile.
update	Advises ant to overwrite files in the already built JAR file.

Continuing our **Hello World** project, let us add a new target to produce the jar files. But before that let us consider the jar task:

```
<jar destfile="${web.dir}/lib/util.jar"
     basedir="${build.dir}/classes"
     includes="faxapp/util/**"
     excludes="**/Test.class"
/>
```

In this example, the **web.dir** property points to the path of the web source files. In our case, this is where the util.jar will be placed.

The **build.dir** property in this example points to the build folder where the class files for the util.jar can be found.

In this example, we create a jar file called **util.jar** using the classes from the **faxapp.util.*** package. However, we are excluding the classes that end with the name Test. The output jar file will be place in the webapp's lib folder.

If we want to make the util.jar an executable jar file we need to add the **manifest** with the **Main-Class** meta attribute.

Therefore the above example will be updated as:

```
<jar destfile="${web.dir}/lib/util.jar"
     basedir="${build.dir}/classes"
     includes="faxapp/util/**"
     excludes="**/Test.class">
     <manifest>
      <attribute name="Main-Class" value="com.tutorialspoint.util.FaxUtil"/>
     </manifest>
</jar>
```

To execute the jar task, wrap it inside a target (most commonly, the build or package target, and run them.

```
<target name="build-jar">
<jar destfile="${web.dir}/lib/util.jar"
     basedir="${build.dir}/classes"
     includes="faxapp/util/**"
     excludes="**/Test.class">
     <manifest>
      <attribute name="Main-Class" value="com.tutorialspoint.util.FaxUtil"/>
     </manifest>
</jar>
</target>
```

Running ant on this file will create the util.jar file for us..

The following outcome is the result of running the ant file:

```
C:\>ant build-jar
Buildfile: C:\build.xml

BUILD SUCCESSFUL
Total time: 1.3 seconds
```

The util.jar file is now placed in the output folder.

Ant - Creating WAR files

Creating WAR files with Ant is extremely simple, and very similar to the creating JAR files task. After all WAR file, like JAR file is just another ZIP file, isn't it?

The WAR task is an extension to the JAR task, but it has some nice additions to manipulate what goes into the WEB-INF/classes folder, and generating the web.xml file. The WAR task is useful to specify a particular layout of the WAR file.

Since the WAR task is an extension of the JAR task, all attributes of the JAR task apply to the WAR task. Below are the extension attributes that are specify to the WAR task:

Attributes	Description
webxml	Path to the web.xml file

lib	A grouping to specify what goes into the WEB-INF\lib folder.
classes	A grouping to specify what goes into the WEB-INF\classes folder.
metainf	Specifies the instructions for generating the MANIFEST.MF file.

Continuing our **Hello World** Fax Application project, let us add a new target to produce the jar files. But before that let us consider the war task. Consider the following example:

```
<war destfile="fax.war" webxml="${web.dir}/web.xml">
   <fileset dir="${web.dir}/WebContent">
      <include name="**/*.*"/>
   </fileset>
   <lib dir="thirdpartyjars">
      <exclude name="portlet.jar"/>
   </lib>
   <classes dir="${build.dir}/web"/>
</war>
```

As per the previous examples, the **web.dir** variable refers to the source web folder, i.e, the folder that contains the JSP, css,javascript files etc.

The **build.dir** variable refers to the output folder - This is where the classes for the WAR package can be found. Typically, the classes will be bundled into the WEB-INF/classes folder of the WAR file.

In this example, we are creating a war file called fax.war. The WEB.XML file is obtained from the web source folder. All files from the 'WebContent' folder under web are copied into the WAR file.

The WEB-INF/lib folder is populated with the jar files from the thirdpartyjars folder. However, we are excluding the portlet.jar as this is already present in the application server's lib folder. Finally, we are copying all classes from the build directory's web folder and putting into the WEB-INF/classes folder.

Wrap the war task inside an Ant target (usually package) and run it. This will create the WAR file in the specified location.

It is entirely possible to nest the classes, lib, metainf and webinf directors so that they live in scattered folders anywhere in the project structure. But best practices suggest that your Web project should have the Web Content structure that is similar to the structure of the WAR file. The Fax Application project has its structure outlined using this basic principle.

To execute the war task, wrap it inside a target (most commonly, the build or package target, and run them.

```
<target name="build-war">
   <war destfile="fax.war" webxml="${web.dir}/web.xml">
   <fileset dir="${web.dir}/WebContent">
      <include name="**/*.*"/>
   </fileset>
   <lib dir="thirdpartyjars">
      <exclude name="portlet.jar"/>
   </lib>
   <classes dir="${build.dir}/web"/>
</war>
</target>
```

Running ant on this file will create the **fax.war** file for us..

The following outcome is the result of running the ant file:

```
C:\>ant build-war
Buildfile: C:\build.xml
```

```
BUILD SUCCESSFUL
Total time: 12.3 seconds
```

The fax.war file is now placed in the output folder. The contents of the war file will be:

```
fax.war:
  +---jsp             This folder contains the jsp files
  +---css             This folder contains the stylesheet files
  +---js              This folder contains the javascript files
  +---images          This folder contains the image files
  +---META-INF        This folder contains the Manifest.Mf
  +---WEB-INF
        +---classes   This folder contains the compiled classes
        +---lib       Third party libraries and the utility jar files
        WEB.xml       Configuration file that defines the WAR package
```

Ant - Executing Java code

You can use Ant to execute java code. In this example below, the java class takes in an argument (administrator's email address) and sends out an email.

```
public class NotifyAdministrator
{
  public static void main(String[] args)
  {
    String email = args[0];
    notifyAdministratorviaEmail(email);
    System.out.println("Administrator "+email+" has been notified");
  }
  public static void notifyAdministratorviaEmail(String email)
  {
      //......
  }
}
```

Here is a simple build that executes this java class.

```
<?xml version="1.0"?>
<project name="sample" basedir="." default="notify">
  <target name="notify">
    <java fork="true" failonerror="yes" classname="NotifyAdministrator">
      <arg line="admin@test.com"/>
    </java>
  </target>
</project>
```

When the build is executed, it produces the following outcome:

```
C:\>ant
Buildfile: C:\build.xml

notify:
     [java] Administrator admin@test.com has been notified

BUILD SUCCESSFUL
Total time: 1 second
```

In this example, the java code does a simple thing - to send an email. We could have used the built in Ant task to do that. However, now that you have got the idea you can extend your build file to call java code that performs complicated things, for example: encrypts your source code.